Debreceni Egyetem

Informatika Kar

Fejlett keresőalgoritmusok

Témavezető:
Kósa Márk Szabolcs
Egyetemi tanársegéd

Készítette:
Andrejcsik Tamás
programtervező informatikus

Debrecen
2010

Tartalomjegyzék

Bevezetés

Mesterséges intelligencia

„Az MI miért nem a matematika egyik ága?A válasz az hogy az MI a kezdetek óta sajátjának tekintette az olyan emberi képességek duplikálását, mint a kreativitás, az önfejlesztés és a nyelv használata.” Stuart Russell, Peter Norvig. Mesterséges intelligencia

A fejlett keresőalgoritmusok a mesterséges intelligencia kutatási területéhez tartoznak így ez a szakdolgozat is ebbe a témakörbe sorolható. A mesterséges intelligencia elnevezés a második világháború után 1956-bem született meg. Már maga az elnevezés is izgalmas. Az emberi intelligencia és az emberi gondolkodás folyamata is hitetlenül összetett aminek a működését régóta próbálják megfejteni. Sokszor érezték már úgy a kutatók, hogy közel járnak a célhoz de végül mindig rá kell döbbenniük hogy mennyire bonyolult terület ez.
A mesterséges intelligencia ahogy azt egy átlag ember is elképzeli még csak a sci-fi világában létezik de a történelem során többször is volt hogy az író által megálmodott fantasztikus történetek valósággá váltak. Jó példa erre az űrutazás.

A kezdetektől fogva sokat vártak a mesterséges intelligencia kutatásától. Természetesen voltak nagy vállalások amik végül nem valósultak meg illetve még váratnak magukra de voltak hatalmas sikerei is.
A sakkvilágbajnokot is képes elverni egy sakkozó gép. Először az IBM által fejlesztett számítógép (Deep Blue) 1997-ben egy szabályos hatjátszmás páros mérkőzésen New Yorkban 3,5-2,5 arányban legyőzte Garri Kaszparovot az akkori sakkozás világbajnokát. Ezt követően az IBM részvényei 18 milliárd dollárral emelkedtek. A gép kifejlesztési költsége körülbelül 20 millió dollár lehetett. Látható, hogy igen jelenetős haszonra tett szert vele a nagy cég.
Természetesen ezt a területet és felhasználja a hadi ipar. 1991-ben az amerikai haderő a szállítások ütemezésére egy Dart (Dinamic Analisis and Replanning Tool) nevű automatikus logisztikai tervező programot fejlesztett ki. A rendszer működése közben 50 ezer jármű szállításával is tudott foglalkozni. A korábban hetekig tartó tervezést néhány óra alatt végezte

el a program és csupán ezzel az egy alkalmazásával megtérítette a Védelmi Kutatási Ügynökség által 30 év alatt MI-re költött pénzt.
Napjainkban van elterjedőben a Google fordító programja. Ez kivételesen nem nyelvtani alapokon fordít hanem statisztikai alapokon működik. Egy öntanuló algoritmust használnak amelynek nagy mennyiségű kétnyelvű szövegre van szüksége és ebből „megtanul” az algoritmus fordítani.
A gyakorlati alkalmazásokat hosszasan lehetne még sorolni. Így már talán látható, hogy nem időpazarlás annak a munkája aki emberi módon gondolkodó (vagy ezt a látszatot keltő) gépek megalkotásán dolgozik.

Célkitűzés

A választásom két okból esett erre a témára. Az első, hogy számomra is mint sok informatikus számára érdekesen cseng az, hogy mesterséges intelligencia és érdeklődve figyelem a hozzá kapcsolódó híreket. A második az egyetemen elvégzett "A mesterséges intelligencia alapjai" című tárgy melynek gyakorlatán a szó szoros értelemben játékosan tanulhattuk meg a mesterséges intelligencia alapjait. Itt egy illetve kétszemélyes játékokat kellett leimplementálni és a tanult keresőalgoritmusokkal ki is próbálhattuk őket. Számomra ez volt a legérdekesebb gyakorlat az egyetemen így ezt a témakört választottam a szakdolgozatom témájának. A szakdolgozatomban az ott elkezdett tanulmányaimat szeretném folytatni. A mesterséges intelligencia alapjai gyakorlaton közösen megírt keretrendszert felhasználva egy általam implementált egyszemélyes játékon keresztül fogom bemutatni a Szimulált lehűtés nevű fejlett keresőalgoritmust és összehasonlítom a gyakorlaton megismert szélességi keresővel. Szakdolgozatomban a szimulált lehűtésen kívül foglalkozok a hegymászó keresővel és a véletlen keresővel is de főleg azért, hogy a szimulált lehűtés működését még inkább megértsük és legyen azt mihez hasonlítani.
A szakdolgozat végére látni fogjuk, hogy hogyan kell egy játékot a számítógép számára is érthetően leírni majd azt implementálni.

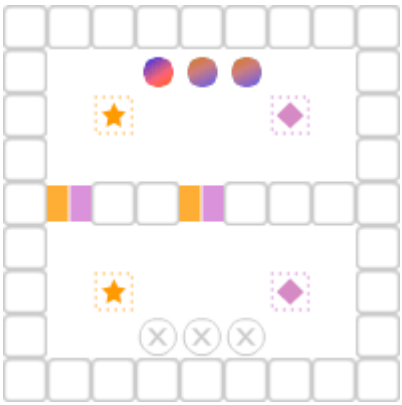
Ezután a keresőalgoritmusokkal ismerkedhetünk meg és ezekre is láthatunk majd egy lehetséges implementációt.
Végül a keresők értékelése következik. Minden keresővel kipróbáltam a játékot és az ott elér eredmények alapján a szokásos szempontok szerint összehasonlítom a keresőket. Ebből láthatjuk majd az egyes keresők előnyeit hátrányit és, hogy milyen helyzetbe melyen keresőt érdemes választani.

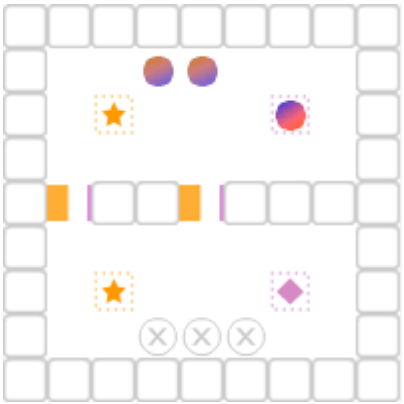
I. fejezet

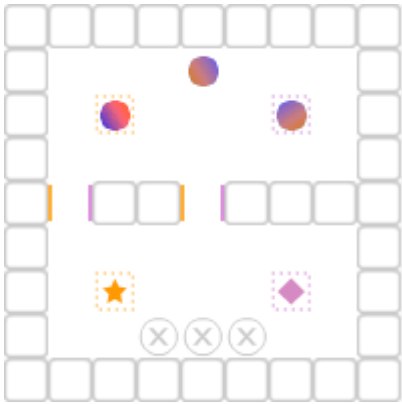
1.1 A játék leírása

Az általam választott játék egy egyszemélyes logikai játék melynek neve Threesome.

Threesome







Három cimbora Attila Márk és Tamás (a képen a 3 színes labda) bennragadtak egy épületben. A kiszabadulásukhoz át kell jutniuk egy olyan szobába ahonnan már van számukra kiút amit most x-szel jelölünk. Az ajtón akkor tud átmenni egyikük ha a másik két fiú az ajtónak megfelelő színű négyzeten áll tehát csak összehangolt mozgással tudnak kijutni. A csapatban Márk a főnök. Vajon sikerülni fog neki megtalálni a helyes utat?

1.2 Állapottér reprezentáció és implementációja

Az imént ismertetett játékleírást egy ember képes megérti. Meg kell viszont értetnünk a szabályokat a számítógéppel is vagyis le kell írni a feladatot, problémát a számítógép nyelvén is. Ehhez először egy logikai formulákat tartalmazó leírást kell készíteni, ez lesz az állapottér reprezentáció. Ennek segítségével készíthetjük el az implementációt valamelyik programozási nyelvben. Én a Java-t használtam. A következőkben az állapottér reprezentációt mutatom be és egyes részeknek a Java megvalósítását is mellékelem. Az állapottér reprezentációnak nem a teljes Java kódját írom le, csak egyes fontosabb részeket amik kifejezetten az állapottér reprezentációnak a megvalósításához köthetők.

1.3 Threesome állapottér reprezentáció és implementáció

Segéd mátrix ami a pályát szimbolizálja: $S = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ n & 0 & 0 & 0 & l \\ F & nl & F & nl & F \\ n & 0 & 0 & 0 & l \\ 0 & f & f & f & 0 \end{pmatrix}$

F=Fal. l=Lila kapcsoló. n=narancssárga kapcsoló. f = Cél (finish).
nl=Ajtó aminek a kinyitásához egy lila és egy narancssárga kapcsoló kell.

Ennek a pályának a Java implementációja.

```
private final static String[][] PÁLYA = { //
    {"0 ", "0 ", "0 ", "0 ", "0 "},
    {"n ", "0 ", "0 ", "0 ", "l "},
    {"F ", "nl", "F ", "nl", "F "},
    {"n ", "0 ", "0 ", "0 ", "l "},
    {"0 ", "f ", "f ", "f ", "0 "},
};
```

1 ábra: A pálya

1.3.1 Alaphalmazok

A játék fontosnak vélt jellemzőit rögzítjük. Ezek a jellemzők egy érték m-est alkotnak. Ezekkel a jellemzőkkel mondhatjuk meg a játékunk épp milyen állapotban van. Jelen esetben a fontos jellemzők a fent megadott mátrix és a 3 barát pozíciója ebben a mátrixban.

$H_{x1} = \{1,2,3,4,5\}$

$H_{x1} = H_{y1} = H_{x2} = H_{y2} = H_{x3} = H_{y3}$ az alaphalmazok

$H_{x1} \times H_{y1} \times H_{x2} \times H_{y2} \times H_{x3} \times H_{y3}$ $\{(1,1,1,2,1,3), (1,1,1,2,1,4), (1,1,1,2,1,5),$

$(1,1,1,2,2,1), (1,1,1,2,2,2), \ldots, (5,2,5,3,5,4), (5,3,5,4,5,5)\}$ //5*5*5*5*5*5=15625 elem.

1.3.2 Állapotok halmaza

A fenti alapállapotok még még figyelmen kívül hagyják az olyan lehetőséget hogy az adott játékos mondjuk olyan értéken áll ahol nem lenne szabad. Pl: fal van az adott pozíción. Azok az állapotok amik eleget tesznek a játék szabályainak lesznek a valódi állapotok. Ezeket a szabályokat a következőképpen fogalmazzuk meg.

A $h=(h_{x1}, h_{y1}, h_{x2}, h_{y2}, h_{x3}, h_{y3}) \in H_{x1} \times H_{y1} \times H_{x2} \times H_{y2} \times H_{x3} \times H_{y3}$ elemhatos a probléma valódi állapota ha teljesíti a következő *kényszerfeltételeket*:

1. A 3 korong 3 különböző mezőn van.
2. Nem állhatnak a korongok falon.
3. Nem lehet egyszerre két korong is ajtóban.

Kényszerfeltétel (h) $\equiv \forall i \forall j \left((h_{xi} = h_{xj} \wedge h_{yi} = h_{yj}) \supset i = j\right) \forall i \left(s_{h_{xi}h_{yi}} \neq F\right) \wedge$

$\forall i \left(s_{h_{xi}h_{yi}} = nl \supset \forall j \left(s_{h_{xj}h_{yj}} = nl \supset i = j\right)\right)$

Valódi állapotok halmazát jelöljük A-val. $A \subset H_{x1} \times H_{y1} \times H_{x2} \times H_{y2} \times H_{x3} \times H_{y3}$
Ezeknek a feltételeknek eleget tesz a kezdő állapot. Az operátorok alkalmazási előfeltételei pedig nem engedi, hogy olyan helyre lépjünk vagy olyan lépést tegyünk ami szabálytalan. Ezért a fenti formulákat az operátorok alkalmazási előfeltételei tartalmazzák. Ezeket később láthatjuk.

1.3.3 Kezdő állapot

Az az állapot miből a játékos indul. Kezdő=(2,1,3,1,4,1) $\in A$

```
public ThreesomeÁllap2() { //Konstruktor. Kezdő állapot

    this.index = new int[6];
    this.index[0] = 1;
    this.index[1] = 0;
    this.index[2] = 2;
    this.index[3] = 0;
    this.index[4] = 3;
    this.index[5] = 0;
}
```

2. ábra: Kezdő állapot

1.3.4 Célállapot

Az az állapot amibe a játékos el szeretne jutni. $C = \{c | c \in A \text{ és Célfeltétel(c)}\}$

Célfeltétel(c)= $\forall i \left(s_{h_{xi} h_{yi}} = f \right)$

```
public boolean célállapote() {
    for (int i = 0; i <= 2; ++i) {
        //[SOR,OSZLOP] az indexelés mátrixnál --- (y,x) az (x,y) helyett
        if (!PÁLYA[this.index[1 + i * 2]][this.index[0 + i * 2]].equals("f "))
            return false;  //equalst kell használni '==' helyett
        }
    }
    return true;
}
```

3. ábra: Célállapot feltétele

1.3.5 Operátorok

A játék közben megtett lépésekre is szabályok vonatkoznak. Ezeket a szabályokat is le kell írni. A lépeseket operátorokkal hajthatjuk végre. Jelen esetben 4 operátor lesz. Ezek a fel, le, jobbra illetve balra való lépések. Hogy egy operátort alkalmazhassunk (megtehessünk egy lépést) eleget kell tennünk a következő szabályoknak.

1.3.6 Operátorok halmaza

$O = \{\text{Fel(oszlop,sor), Le(oszlop,sor), Jobbra(oszlop,sor), Balra(oszlop,sor)}\}$

Az operátorok alkalmazhatóak egy $h=(h_{x1} , h_{y1} , h_{x2} , h_{y2} , h_{x3} , h_{y3}) \in A$ állapotra ha teljesítik a következő *alkalmazhatósági előfeltételeket*:

1. Korong legyen ott ahonnan lépni akarunk.
2. Nem léphetünk falra.
3. Nem léphetünk másik labdára.
4. Nem lépünk le a pályáról (az indexek 1..5 határon belül maradnak).
5. Ha ajtóra akarunk lépni a másik két korong két olyan kapcsolón álljon ami az ajtót kinyitja.

```
//Nem alkalmazható az Op ha falra akarunk lépni
if (PÁLYA[sorY + sor][oszlopX + osz].equals("F ")) {
    //System.out.println("Falra akarsz lépni pedig arra nem lehet.");
    return false;
}

//Ha a labdával egy másik labdára akarunk lépni
for (int i = 0; i <= 2; ++i) {
    if ((oszlopX + osz == this.index[0 + i * 2]) &&
            (sorY + sor == this.index[1 + i * 2])) {
        //System.out.println("Egy másik labda áll ott ahova lépni akarsz.");
        return false;
    }
}
```

4. ábra: Ha falra vagy egy másik labdára akarunk lépni

```
 /Ha ajtóra akar lépni
if (PÁLYA[sorY + sor][oszlopX + osz].equals("nl")) {
    switch (melyikLabda) {
        case 0:
            if (!(PÁLYA[this.index[3]][this.index[2]].equals("n ") &&
                    PÁLYA[this.index[5]][this.index[4]].equals("l ") ||
                    PÁLYA[this.index[3]][this.index[2]].equals("l ") &&
                    PÁLYA[this.index[5]][this.index[4]].equals("n "))) {
                return false;
            }
            break;
        case 1:
            if (!(PÁLYA[this.index[1]][this.index[0]].equals("n ") &&
                    PÁLYA[this.index[5]][this.index[4]].equals("l ") ||
                    PÁLYA[this.index[1]][this.index[0]].equals("l ") &&
                    PÁLYA[this.index[5]][this.index[4]].equals("n "))) {
                return false;
            }
            break;
        case 2:
            if (!(PÁLYA[this.index[1]][this.index[0]].equals("n ") &&
                    PÁLYA[this.index[3]][this.index[2]].equals("l ") ||
                    PÁLYA[this.index[1]][this.index[0]].equals("l ") &&
                    PÁLYA[this.index[3]][this.index[2]].equals("n "))) {
                return false;
            }
            break;
    }
}
```

5. ábra: Átjáróra lépés feltétele

- Fel előfeltétel(h) ≡

$$\exists i\left(oszlop = \mathrm{h}_{\mathrm{xi}} \wedge sor = h_{yi}\right) \wedge \left(\mathrm{s}_{\mathrm{oszlop,\ sor\ -1}} \neq F\right) \wedge \neg \exists i\left(h_{xi} = oszlop \wedge h_{yi} = sor - 1\right) \wedge$$

$$\left(sor \geq 2\right) \wedge \left(s_{oszlop\,,sor-1} = nl\right) \supset \exists i \exists j\left(s_{h_{xi}h_{yi}} = n \wedge s_{h_{xj}h_{yj}} = l\right)$$

- Le előfeltétel(h) ≡

$$\exists i\left(oszlop = \mathrm{h}_{\mathrm{xi}} \wedge sor = h_{yi}\right) \wedge \left(\mathrm{s}_{\mathrm{oszlop,\ sor\ +1}} \neq F\right) \wedge \neg \exists i\left(h_{xi} = oszlop \wedge h_{yi} = sor + 1\right) \wedge$$

$$\left(sor \leq 4\right) \wedge \left(s_{oszlop\,,sor+1} = nl\right) \supset \exists i \exists j\left(s_{h_{xi}h_{yi}} = n \wedge s_{h_{xj}h_{yj}} = l\right)$$

- Jobbra előfeltétel(h) ≡

$$\exists i\left(oszlop = \mathrm{h}_{\mathrm{xi}} \wedge sor = h_{yi}\right) \wedge \left(\mathrm{s}_{\mathrm{oszlop\ +1,\ sor}} \neq F\right) \wedge \neg \exists i\left(h_{xi} = oszlop + 1 \wedge h_{yi} = sor\right) \wedge$$

$$\left(oszlop \leq 4\right) \wedge \left(s_{oszlop+1,sor} = nl\right) \supset \exists i \exists j\left(s_{h_{xi}h_{yi}} = n \wedge s_{h_{xj}h_{yj}} = l\right)$$

- Balra előfeltétel(h) ≡

$$\exists i\left(oszlop = \mathrm{h}_{\mathrm{xi}} \wedge sor = h_{yi}\right) \wedge \left(\mathrm{s}_{\mathrm{oszlop\ -1,\ sor}} \neq F\right) \wedge \neg \exists i\left(h_{xi} = oszlop - 1 \wedge h_{yi} = sor\right) \wedge$$

$$\left(oszlop \geq 2\right) \wedge \left(s_{oszlop-1,sor} = nl\right) \supset \exists i \exists j\left(s_{h_{xi}h_{yi}} = n \wedge s_{h_{xj}h_{yj}} = l\right)$$

Az operátorokat alkalmazva egy h=(h_{x1} , h_{y1} , h_{x2} , h_{y2} , h_{x3} , h_{y3}) ∈ A állapotra a következőképpen definiált h'=(h_{x1}', h_{y1}', h_{x2}', h_{y2}', h_{x3}', h_{y3}') ∈ A elemhatost kapjuk:

A. Ha h_{xi}=oszlop és h_{yi}=sor:
   1. Fel operátor esetén: h_{xi}'=oszlop és h_{yi}'=sor-1
   2. Le operátor esetén: h_{xi}'=oszlop és h_{yi}'=sor+1
   3. Jobbra operátor esetén: h_{xi}'=oszlop+1 és h_{yi}'=sor
   4. Balra operátor esetén: h_{xi}'=oszlop-1 és h_{yi}'=sor

B. Különben: h_{xi}'=h_{xi} és h_{yi}'=h_{yi}

```java
import állapottérrep.Operátor;

public class JobbraOper extends Operátor{

    private int oszlopX;
    private int sorY;

    public JobbraOper(int oszlop, int sor) {
        this.oszlopX = oszlop;
        this.sorY = sor;
    }

    public int getOszlopX() {
        return oszlopX;
    }

    public int getSorY() {
        return sorY;
    }

    @Override
    public String toString() {
        return ("Jobbra operátor(" + this.oszlopX + "," + this.sorY + ")");
    }
}
```

6. ábra: Jobbra operátor

A négy operátor nagyon hasonlít egymásra ezért itt csak a Jobbra operátor kódját ismertetem.

Így megadtuk a probléma állapottér-reprezentációját: p=(A,kezdő,C,O)

1.3.7 Heurisztika

A heurisztika valamilyen plusz ismeret aminek segítségével egy állapotot tudunk értékelni, pontozni. Így egy állapotból ha több állapotba is eljuthatunk a heurisztika segítségével tudjuk eldönteni hogy melyiket válasszuk ,melyik a legjobb.

```
public double Heurisztika() {
    double h = 60;
    h = h + abs((4 - this.index[1])) + abs(1 - this.index[0]);
    h = h + abs((4 - this.index[3])) + abs(3 - this.index[2]);
    h = h + abs((4 - this.index[5])) + abs(2 - this.index[4]);
    if (PÁLYA[this.index[1]][this.index[0]].equals("n ") ||
            PÁLYA[this.index[1]][this.index[0]].equals("l ")) {
        h = h - 10;
    }
    if (PÁLYA[this.index[3]][this.index[2]].equals("n ") ||
            PÁLYA[this.index[3]][this.index[2]].equals("l ")) {
        h = h - 10;
    }
    if (PÁLYA[this.index[5]][this.index[4]].equals("n ") ||
            PÁLYA[this.index[5]][this.index[4]].equals("l ")) {
        h = h - 10;
    }
    if (((PÁLYA[index[1]][index[0]].equals("n ") || PÁLYA[index[1]][index[0]].equals("l ")) &&
            (PÁLYA[index[3]][index[2]].equals("n ") || PÁLYA[index[3]][index[2]].equals("l "))) ||
            ((PÁLYA[index[1]][index[0]].equals("n ") || PÁLYA[index[1]][index[0]].equals("l ")) &&
            (PÁLYA[index[5]][index[4]].equals("n ") || PÁLYA[index[5]][index[4]].equals("l "))) ||
            ((PÁLYA[index[5]][index[4]].equals("n ") || PÁLYA[index[5]][index[4]].equals("l ")) &&
            (PÁLYA[index[3]][index[2]].equals("n ") || PÁLYA[index[3]][index[2]].equals("l ")))) {
        h = h - 10;
    }
    if (index[1] > 2 && index[3] > 2 && index[5] > 2) {
        if (PÁLYA[index[1]][index[0]].equals("f ")) {
            h = h - 20;
        }
        if (PÁLYA[index[3]][index[2]].equals("f ")) {
            h = h - 20;
        }
        if (PÁLYA[index[5]][index[4]].equals("f ")) {
            h = h - 20;
        }
    }
    return h;
```

7. ábra: Heurisztika

1.3.8 Állapottér gráf Kezdő állapot (2,1,3,1,4,1)

Balra(2,1) Jobbra(1,1) Le(2,1) Fel(2,2) Le(3,1) Fel(3,2) Le(4,1) Fel(4,2) Jobbra(4,1) Balra(5,1)

(1,1,3,1,4,1) (2,2,3,1,4,1) (2,1,3,2,4,1) (2,1,3,1,4,2) (2,1,3,1,5,1)

Le(1,1) Fel(1,2)

(1,2,3,1,4,1)

Jobbra(4,1) Balra(5,1)

(1,2,3,1,5,1)

Le(5,1) Fel(5,2)

(1,2,3,1,5,2) (3,5,2,5,4,5)

Le(3,1) Fel(3,2) Le(4,4) Fel(4,5)

(1,2,3,2,5,2). (3,5,2,5,4,4)

Balra(3,2)Jobbra(2,2)b Balra(5,4) Jobbra(4,4)

(1,2,2,2,5,2). (3,5,2,5,5,4)

Le(2,2) Fel(2,3) Le(2,4) Fel(2,5)

(1,2,2,3,5,2) (3,5,2,4,5,4)

Le(2,3) Fel(2,4) Jobbra(1,4) Balra(2,4)

(1,2,2,4,5,2) (3,5,1,4,5,4)

Balra(2,4) Jobbra(1,4) Jobbra(2,5) Balra(3,5)

(1,2,1,4,5,2) (2,5,1,4,5,4)

Balra(5,2) Jobbra(4,2) Le(2,4) Fel(2,5)

(1,2,1,4,4,2). (2,4,1,4,5,4)

Le(4,2) Fel(4,3) Le(2,3) Fel(2,4)

(1,2,1,4,4,3) (2,3,1,4,5,4)

Le(4,3) Fel(4,4) Le(2,2) Fel(2,3)

(1,2,1,4,4,4) (2,2,1,4,5,4)

Jobbra(4,4) Balra(5,4) (1,2,1,4,5,4) Jobbra(1,2) Balra(2,2)

II. fejezet

2 Keresőalgoritmusok

A megoldás keresés pontosan azt a folyamatot takarja amikor is a kezdő állapotból egy lehetséges utat keresünk egy célállapotba úgy, hogy közben betartjuk a játék szabályait. Most, hogy már a számítógép is érti a szabályokat (az előző fejezetben megadtuk az állapottér reprezentációt) elkezdődhet a játék. Ilyenkor az ember leül és elkezd gondolkodni, hogy mit is lépjen, melyik lépés vinnék közelebb őt a célhoz. A gép a megoldás kereséséhez megoldást kereső algoritmusokat használ. Ezekkel „gondolkodik”. Ez a fejezet ezekkel az algoritmusokkal fog foglalkozni. Konkrétan két egymástól nagyon különböző algoritmussal, a szélességi keresővel és a Szimulált lehűtéssel.

2.1 Keresőalgoritmusok fajtái

A keresőalgoritmusok többféle képen lehet csoportosítani. Egyik csoportosítási szempont lehet az, hogy a kereső rendelkezik-e a játékot definiáló leíráson kívül más, plusz információval vagy sem. Ha rendelkezik akkor **informált azaz heurisztikus** algoritmusról van szó. Ha nem akkor pedig **nem informált vagyis szisztematikus** algoritmusról.

Egy másik szempont lehet az, hogy a keresés során milyen adatokat tárolunk az keresésről. Ilyen lehet például a már megvizsgált állapotok listája is vagy az, hogy használ-e a kereső **keresési fát** amit a kezdeti állapotból az állapotátmenet függvényekkel generál. Ha nem tart nyilván ilyen információkat csak egyetlen egy állapotot akkor **lokális keresőalgoritmusról** beszélünk.

Az általam vizsgát két algoritmus minkét szempontban eltérnek egymástól.

2.2 Szélességi keresés (breadth-first search)

Ez egy egyszerű keresési stratégia. A keresési gráf első szintjét a kezdőállapot alkotja. Ebből előállítjuk az összes előállítható állapotot és ezek lesznek a kezdő állapot gyerekei azaz a gráf második szintje. Ezután előállítjuk a második szint állapotaiból az összes előállítható állapotot és ezek lesznek a harmadik szint. A harmadik szint állapotaiból megint előállítjuk az összes előállítható állapotot és ez lesz a negyedik szint és így tovább eddig amíg elő nem áll a célállapot vagy valamilyen feltétel megállítja a keresést. Például egy időkorlát.

```
package kereső.gráffalkereső.szisztematikus;

import kereső.gráffalkereső.GráfbanKereső;
import kereső.Csúcs;
import állapottérrep.Operátor;
import állapottérrep.Állapot;

public class SzélességiKereső extends GráfbanKereső {

    public SzélességiKereső(Állapot kezdőÁll) {  /*Konstruktor*/
        super();
        nyiltak.add(new Csúcs(kezdőÁll));  /*Betenni a csúcsot a nyiltak közé.*/
    }

    public SzélességiKereső(Állapot kezdőÁll, int jellemző) {  /*Konstruktor*/
        super(jellemző);
        nyiltak.add(new Csúcs(kezdőÁll));  /*Betenni a csúcsot a nyiltak közé.*/
    }

    @Override
    public void Keres() {
        System.out.println("Szélességi kereső indul. Adatai:");
        if (összesCélÁllapKelle) {
            System.out.println("  -Az összes célállapotba keresek utat.");
        } else {
            System.out.println("  -Csak egy célállapotba keresek utat.");
        }
        if (utKelle) {
            System.out.println("  -Kiírom a megoldáshoz vezető utat.");
        } else {
            System.out.println("  -Nem írom ki a megoldáshoz vezető utat.");
        }
```

8. ábra: Szélességi kereső

```
while (true) {
    if (nyiltak.isEmpty()) {
        System.out.println("A nyiltak halmaza üres. Vége a keresésnek.");
        break;
    }
    Csúcs aktuálisCsúcs = nyiltak.get(0);  /*Kiválasztás. A lista legelső eleme.*/
    if (aktuálisCsúcs.getAktÁllap().célállapote()) {
        if (this.összesCélÁllapKelle) {
            if (!this.termCsúcsok.contains(aktuálisCsúcs)) {
                this.termCsúcsok.add(aktuálisCsúcs);
            }
            zártak.add(aktuálisCsúcs);
            nyiltak.remove(0);  /*A célállapotot tartalmazó csúcsot már
                                  nem kell kiterjeszteni, átkerül a zártakhoz*/
            continue;
        } else {
            this.termCsúcsok.add(aktuálisCsúcs);
            break;
        }
    }
    for (Operátor op : Állapot.getOpok()) {  /*KITERJESZTÉS*/
        if (aktuálisCsúcs.getAktÁllap().alkalmazhatóe(op)) {
            Csúcs uj = new Csúcs(aktuálisCsúcs, op);
            if (!nyiltak.contains(uj) && !zártak.contains(uj)) {
                nyiltak.add(uj);
            }
        }
    }
    zártak.add(aktuálisCsúcs);  /*A kiterjesztett csúcsot a zártak közé tesszük.*/
    nyiltak.remove(0);  /*A kiterjesztett csúcsot kivesszül a nyiltakból.*/
}
```

9. ábra: Szélességi kereső folytatása

A szélességi kereső nem informált a játék definícióján kívül nem ismer semmilyen plusz információt amivel értékelni tudná az állapototokat, hogy egyik ígéretessebbe-e mint a másik. Gráfban kereső algoritmus tehát sok adatot tárol a memóriában többek között a kezdő állapotból a célállapotba vezető utat.

A szélességi kereső megoldása. A kezdő állapotból 24 lépéssel jut el el a célállapotba. Az alábbi képen a kezdő állapotból vezető út látható. Az első oszlop az első 5 lépés a második oszlop a második 5 és így tovább. Utolsó állapot a célállapot.

```
**********************************************************************
Threesome2 megoldás keresése szélességi megoldáskeresővel.
**********************************************************************
Szélességi kereső indul. Adatai:
  -Csak egy célállapotba keresek utat.
  -Kiírom a megoldáshoz vezető utat.
Megoldások száma: 1
Kezdö állapot:
Állapot(1,0,2,0,3,0) Állapot(0,1,2,1,4,1) Állapot(0,1,3,3,2,1) Állapot(1,1,4,3,1,3) Állapot(1,2,4,4,1,4)
0 @ @ @ 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0
n 0 0 0 1            @ 0 @ 0 @            @ 0 @ 0 1            n @ 0 0 1            n 0 0 0 1
F n1F n1F            F n1F n1F            F n1F n1F            F n1F n1F            F @ F n1F
n 0 0 0 1            n 0 0 0 1            n 0 0 @ 1            n @ 0 0 @            n 0 0 0 1
0 f f f 0            0 f f f 0            0 f f f 0            0 f f f 0            0 @ f f @

Állapot(1,0,2,1,3,0) Állapot(0,1,3,1,4,1) Állapot(0,1,4,3,2,1) Állapot(1,1,4,3,0,3) Állapot(1,2,4,4,2,4)
0 @ 0 @ 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0
n 0 @ 0 1            @ 0 0 @ @            @ 0 @ 0 1            n @ 0 0 1            n 0 0 0 1
F n1F n1F            F n1F n1F            F n1F n1F            F n1F n1F            F @ F n1F
n 0 0 0 1            n 0 0 0 1            n 0 0 0 @            @ 0 0 0 @            n 0 0 0 1
0 f f f 0            0 f f f 0            0 f f f 0            0 f f f 0            0 f @ f @

Állapot(1,1,2,1,3,0) Állapot(0,1,3,2,4,1) Állapot(0,1,4,3,1,1) Állapot(1,2,4,3,0,3) Állapot(1,2,3,4,2,4)
0 0 0 @ 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0
n @ @ 0 1            @ 0 0 0 @            @ @ 0 0 1            n 0 0 0 1            n 0 0 0 1
F n1F n1F            F n1F @ F            F n1F n1F            F @ F n1F            F @ F n1F
n 0 0 0 1            n 0 0 0 1            n 0 0 0 @            @ 0 0 0 @            n 0 0 0 1
0 f f f 0            0 f f f 0            0 f f f 0            0 f f f 0            0 f @ @ 0

Állapot(0,1,2,1,3,0) Állapot(0,1,3,2,3,1) Állapot(0,1,4,3,1,2) Állapot(1,2,4,4,0,3) Állapot(1,3,3,4,2,4)
0 0 0 @ 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0
@ 0 @ 0 1            @ 0 0 @ 1            @ 0 0 0 1            n 0 0 0 1            n 0 0 0 1
F n1F n1F            F n1F @ F            F @ F n1F            F @ F n1F            F n1F n1F
n 0 0 0 1            n 0 0 0 1            n 0 0 0 @            @ 0 0 0 1            n @ 0 0 1
0 f f f 0            0 f f f 0            0 f f f 0            0 f f f @            0 f @ @ 0

Állapot(0,1,2,1,4,0) Állapot(0,1,3,2,2,1) Állapot(1,1,4,3,1,2) Állapot(1,2,4,4,0,4) Állapot(1,4,3,4,2,4)
0 0 0 0 @            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0            0 0 0 0 0
@ 0 @ 0 1            @ 0 @ 0 1            n @ 0 0 1            n 0 0 0 1            n 0 0 0 1
F n1F n1F            F n1F @ F            F @ F n1F            F @ F n1F            F n1F n1F
n 0 0 0 1            n 0 0 0 1            n 0 0 0 @            n 0 0 0 1            n 0 0 0 1
0 f f f 0            0 f f f 0            0 f f f 0            @ f f f @            0 @ @ @ 0
```

10. ábra: Kimenet: A szélességi kereső megoldása

2.3 Véletlen keresőalgoritmus

A szakdolgozat fő témája szimulált lehűtés kereső de ennek megismeréséhez érdemes megnézni a véletlen kereső és a hegymászó algoritmusokat. A véletlen kereső algoritmus nem használ semmilyen heurisztikát és csak egy állapotot tart nyilván (tehát lokális kereső). Működése nagyon egyszerű teljesen véletlenszerűen lépked addig míg végül célállapotba nem ér. Lehet, hogy talál megoldást de hihetetlenül nem hatékony. El lehet képzelni ha valaki úgy akar kirakni, megoldani egy logikai játékot, hogy csak véletlenszerűn lépked össze vissza. El fog tartani egy ideig.

```
java.util.Random random = new java.util.Random();
ArrayList<Operátor> opok;
opok = new ArrayList<Operátor>(Állapot.getOpok());
int opokSzáma = Állapot.getOpok().size();

while (true) {
    if (this.aktCsúcs.getAktÁllap().célállapote()) {
        System.out.println("Megvan a célállapot. ");
        System.out.println("Célbavezető út hossza: " + this.célbaVezetőÚt.size());
        System.out.println("Különböző állapotok száma: " + this.külön.size());
        this.termCsúcsok.add(this.aktCsúcs);
        break;
    }
    int kisorsoltOp = random.nextInt(opokSzáma);
    if (this.aktCsúcs.getAktÁllap().alkalmazhatóe(opok.get(kisorsoltOp))) {
        Csúcs uj = new Csúcs(this.aktCsúcs, opok.get(kisorsoltOp));
        this.célbaVezetőÚt.add(uj);
        if(!this.külön.contains(uj)){
                this.külön.add(uj);
            }
        this.aktCsúcs = uj;
    }
}
```

11. ábra: Véletlen kereső

2.4 Hegymászó keresőalgoritmus

A hegymászó algoritmus is lokális kereső de heurisztikus. Csak és kizárólag olyan állapotba lép tovább ami jobb mint az előző. Nagy hátránya, hogy ha egy lokális maximumba ér az állapottérgráfon akkor nem tud továbbhaladni egyszerűen megáll mivel csak jobb állapotra akar lépni de olyan abban a pontban már nincs. Gyakorlatban ez a kereső nem fordul elő. Ez inkább csak egy alapötlet amit felhasználnak más keresők is mint például a szimulált lehűtés.

```
while (true) {
    if (this.aktCsúcs.getAktÁllap().célállapote()) {
        System.out.println("Megvan a célállapot. ");
        System.out.println("Célbavezető út hossza: " + this.célbaVezetőÚt.size());
        this.termCsúcsok.add(this.aktCsúcs);
        break;
    }
    boolean felt = false;
    for (Operátor op : Állapot.getOpok()) {
        if (this.aktCsúcs.getAktÁllap().alkalmazhatóe(op)) {
            HegymászóCsúcs uj = new HegymászóCsúcs(this.aktCsúcs, op);
            if (this.következőCsúcs.heurisztika > uj.heurisztika) {
                felt = true;
                this.következőCsúcs = uj;
            }
        }
    }
    if (felt) {
        this.aktCsúcs = this.következőCsúcs;
        this.célbaVezetőÚt.add(aktCsúcs);
    } else {
        break;
    }
}
```

12. ábra: Hegymászó kereső

.

2.5 Szimulált lehűtés

Ez egy informált tehát heurisztikát használó lokáli kereső algoritmus. A lokális keresés megértését segíti az **állapottérfelszín**. Képzeljünk el egy olyan koordináta rendszer ahol az x, y síkban az állapotok vannak és minden állapothoz tartozik egy érték amit a heurisztika határoz meg. Ezek a pontok alkotják az állapottérfelszínt aminek a globális minimumába vagy maximumába szeretnék eljutni a feladattól függően.
A szimulált lehűtés a hegymászó algoritmus és a véletlen keresési algoritmus keveréke.
A véletlen keresőhöz hasonlóan véletlenül választja ki a következő lépést de a hegymászóhoz hasonlóan olyan állapot felé törekszik ami jobb mint az előző de nem feltétlen a legjobbat választja.

```
function SZIMULÁLT-LEHŰTÉS(probléma, lehűtési terv) returns egy megoldási állapot
    inputs: probléma, egy probléma
            lehűtési terv, egy leképzés időről „hőmérsékletre"
    local variables: aktuális, egy csomópont
                     következő, egy csomópont
                     T, egy „hőmérséklet", ami a lefelé lépések valószínűségét szabályozza

    aktuális ← CSOMÓPONTOT-LÉTREHOZ(KIINDULÓ-ÁLLAPOT[probléma])
    for t ← 1 to ∞ do
        T ← lehűtési terv[t]
        if T = 0 then return aktuális
        következő ← az aktuális egy véletlenszerűen kiválasztott követő csomópontja
        ΔE = ÉRTÉK[következő] – ÉRTÉK[aktuális]
        if ΔE > 0 then aktuális ← következő
        else aktuális ← következő csak e^(−ΔE/T) valószínűséggel
```

13. ábra: Szimulált lehűtés keresési algoritmusa

Ez úgy lehetséges, hogy a véletlenül kiválasztott következő lépést biztosan megteszi ha az jobb állapotba viszi (mint egy hegymászó) viszont egy bizonyos valószínűséggel a rosszabb állapotba haladó lépést is meglépi.
Ezzel lehetőséget adva arra, hogy kikerüljön az állapottérfelszín lokális maximumából amire ugye a hegymászó algoritmus nem képes. Az a bizonyos valószínűség pedig két dologtól függ. Az egyik, hogy mennyivel rosszabb állapotba akar lépni. Minél rosszabb annál kisebb a valószínűsége hogy megteszi a lépést. A másik pedig egy időfüggvény aminek az a feladata, hogy az idő elteltével egyre kisebb eséllyel engedje a rossz lépések megtételét. Honnan jött z időfüggvény ötlete és miért hívják időfüggvénynek?
A válasz a szimulált lehűtést ihlető hűtési eljárásban keresendő. Ezt az algoritmust ugyanis a kohászatban ismert lehűtési eljárás ihlette, innen is ered a neve. Ezt az eljárást arra használják hogy az adott anyagot alapállapotba hozzák. Az alapállapot azt jelenti hogy az anyagi részecskék jól struktúrált kristályrácsba rendeződnek. Ezt úgy érik el, hogy felmelegítik az anyagot annyira, hogy a részecskék szabadon mozoghassanak (olvadásig) és fokozatosan lehűtik úgy, hogy a részecskék az alapállapot szerint rendeződjenek el. A művelet nem lesz sikeres ha az anyagot nem melegítjük fel eléggé vagy ha a lehűtés nem elég fokozatos.
Az algoritmust az időfüggvény segítségével lehet „hangolni” ugyanis ezt az algoritmust nem lehet teljesen általánosan megírni. Mint ahogy a valóságban is a különböző anyagokat más hőmérsékletre kell felhevíteni aztán más ütemben lehűteni, az algoritmust is hangolni kell az egyes problémákhoz illetve azok heurisztikájához.

2.5.1A szimulált lehűtés implementációja

```
public class SzimuláltLehűtés extends Kereső {

    protected SzimuláltLehűtésCsúcs aktCsúcs;
    protected List<Csúcs> célbaVezetőÚt;
    { célbaVezetőÚt = new LinkedList<Csúcs>(); }
    public SzimuláltLehűtés(Állapot kezdőÁll) {  //Konstruktor
        super(); //Nem fontos beírni ez alapértelmezett.
        célbaVezetőÚt.add(new Csúcs(kezdőÁll));
        this.aktCsúcs = new SzimuláltLehűtésCsúcs(kezdőÁll);
    }
    public SzimuláltLehűtés(Állapot kezdőÁll, int jellemző) {  //Konstruktor
        super(jellemző);
        célbaVezetőÚt.add(new Csúcs(kezdőÁll));
        this.aktCsúcs = new SzimuláltLehűtésCsúcs(kezdőÁll);
    }
    @Override
    public void Keres() { //throw new UnsupportedOperationException("Not support
        System.out.println("Szimulált lehűtés kereső indul. Adatai:");
        if (összesCélÁllapKelle) {
            System.out.println("  -Az összes célállapotba keresek utat.");
        } else {
            System.out.println("  -Csak egy célállapotba keresek utat.");}
        if (utKelle) {
            System.out.println("  -Kiírom a megoldáshoz vezető utat.");
        } else {
            System.out.println("  -Nem írom ki a megoldáshoz vezető utat."); }
        java.util.Random random = new java.util.Random();
        ArrayList<Operátor> opok;
        opok = new ArrayList<Operátor>(Állapot.getOpok());
        int opokSzáma= Állapot.getOpok().size();
        double seged = 18;
        double idő = 0;
        double delta = 0;
```

14. ábra: Szimulált lehűtés

```
while (true) {
    if (this.aktCsúcs.getAktÁllap().célállapote()) {
        System.out.println("Megvan a célállapot.");
        System.out.println("Célbavezető út hossza: " +
                this.célbaVezetőÚt.size());
        this.termCsúcsok.add(this.aktCsúcs);
        break;
    }
    int kisorsoltOp = random.nextInt(opokSzáma);
    if (this.aktCsúcs.getAktÁllap().alkalmazhatóe(opok.get(kisorsoltOp))) {
        SzimuláltLehűtésCsúcs uj = new
                SzimuláltLehűtésCsúcs(this.aktCsúcs, opok.get(kisorsoltOp));
        delta = this.aktCsúcs.heurisztika - uj.heurisztika;
        if (delta >= 0) {
            seged = seged - (seged / 100);
            this.célbaVezetőÚt.add(uj);
            this.aktCsúcs = uj;
        } else {
            int valószínűség = random.nextInt(100);
            idő = seged + 1;
            double x = delta / (idő);
            x = Math.pow(Math.E, x);
            x = (x * 100) + 1;
            if (valószínűség < x) {
                seged = seged - (seged / 100);
                this.célbaVezetőÚt.add(uj);
                this.aktCsúcs = uj;
            }
        }
    }
}
```

15. ábra: Szimulált lehűtés folytatása

III. Fejezet

3 A keresőalgoritmusok hatékonyságmérése

A keresőalgoritmusokat hatékonyságát 4 szempont alapján mérik. Ezek a következők.
Teljesség: a keresőalgoritmus garantáltan megtalálja-e a megoldást amennyiben az létezik?
Optimalitás: a keresőalgoritmus a legoptimálisabb megoldást találja-e meg?
Időigény: megoldás megtalálásának ideje?
Tárigény: a keresés során mennyi memóriára van szükség?

Nézzük hát a fent ismertetett 3 keresőalgoritmus értékelését általánosságban, ezt követően pedig a gyakorlatban. A kereső algoritmusok gyakorlati működését a fenti játékon produkált eredményeik alapján vizsgálom. A véletlen és a szimulált lehűtés keresők eredményeiből készítettem statisztikát. 100-szor futtattam a két keresőt és elmentettem az megoldás megtalálásához szükséges időt, a megtalált megoldás hosszát vagy lépésszámát és a keresés során bejárt állapotok számát (összesen 22*21*20=9240 állapot lehetséges). Ezekből diagrammot készítettem amin látható az egyes értékek közötti összefüggés vagy az eltérés.

3.1 Teljesség

A **szélességi kereső** teljes tehát ha létezik megoldás akkor azt biztos megtalálja. Ha nem létezik megoldás akkor azt véges reprezentációs gráfban a gráf teljes bejárásával felismeri de végtelen gráfban a végtelenségig (vagy mondjuk a memória elfogyásáig) keresne.
A Threesome játéknak ezen a pályán van megoldása és ezt meg is találta a kereső. Fentebb a 10. ábrán látható a kereső kimenete.

A **hegymászó algoritmus** nem teljes mivel ha létezik is megoldás ő gyakran megáll egy globális maximumba ahonnan nem tud továbbmenni pedig néhány rosszabb állapot után lehet, hogy az addiginál még jobb állapotot találna. Ezért is született meg a szimulált lehűtés.
A Threesome játékban is megrekedt és nem talál megoldást.

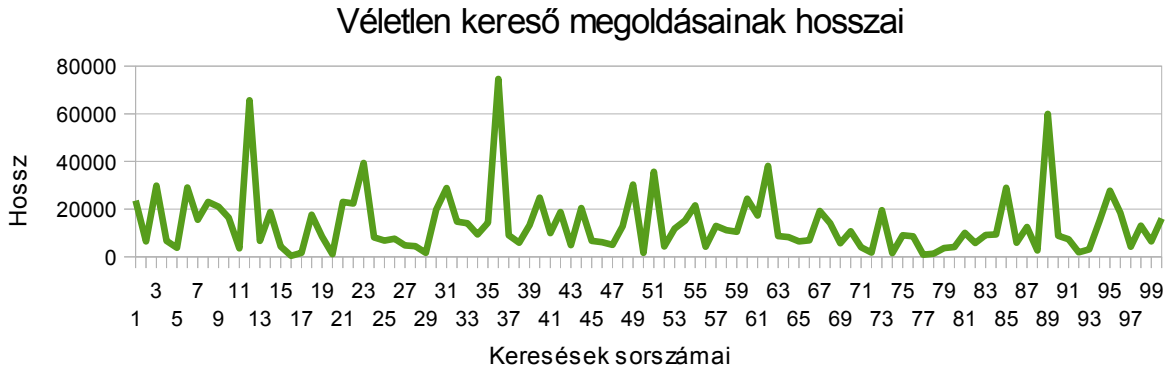
A **véletlen algoritmus** teljes hisz bármerre továbbhaladhat véletlenszerűen és a célba is eljut előbb utóbb. De inkább csak utóbb. Később látni fogjuk, hogy mennyire nem hatékony ez a kereső. A fenti játékban is mindig megtalálta a megoldást de nagyon változó idő alatt.

És végül a **szimulált lehűtés**. Ez az algoritmus nem teljes viszont bizonyítható, hogy az időfüggvény kellően lassú csökkentésével 1-hez tartó valószínűséggel a célban fog megállni. Tesztelés során azt tapasztaltam hogyha az időfüggvény túl gyorsan csökken megreked egy lokális maximumba ha pedig túl lassan annál inkább véletlenszerű keresőhős hasonló eredményt produkál azaz sok ideig barangol össze vissza. A jelenlegi időfüggvénnyel minden tesztelés során talált megoldást.

3.2 Optimalitás

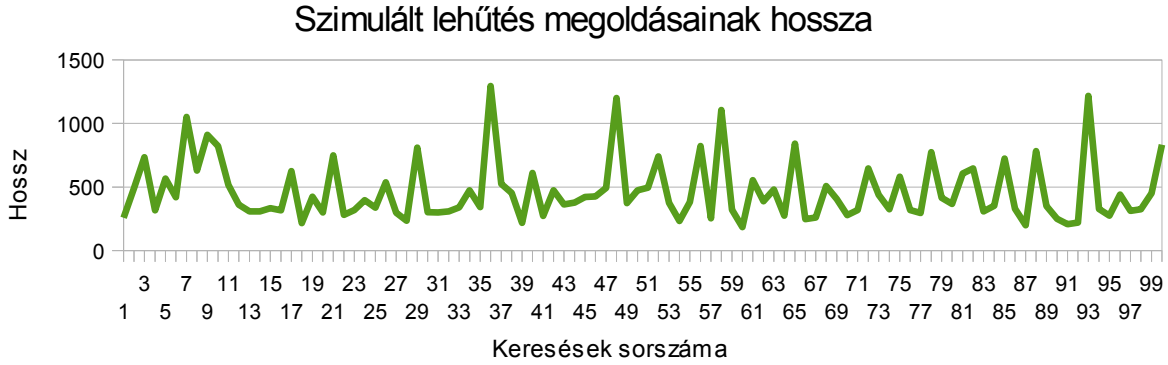
A véletlen és szimulált lehűtés összehasonlításával kezdek.

A **véletlen kereső** az optimális megoldástól nagyon távol van. Elvileg van rá esély, hogy véletlenül az optimális megoldást produkáló operátorsorozatot sorsolja ki de gyakorlatilag ez esélytelen. Hogy mennyire nem hatékony ez a kereső?



16. ábra: Véletlen kereső megoldásainak hossza

A **szimulált lehűtés** sem optimális megoldást ad. Itt összehasonlítható a véletlen keresővel, hogy mennyivel is hatékonyabb ez az algoritmus.



17. ábra: Szimulált lehűtés megoldásainak hossza

Az optimális megoldás 24 lépés alatt ér a célba. Ehhez képest elég rossznak tűnik még a szimulált lehűtés eredménye is.

Megoldás hossza

Szimulált		Véletlen
466,48	**ÁTLAG**	13729,48
183	**MIN**	421
1295	**MAX**	74597

Látható hogy még a szimulált lehűtés legjobb találata is többszöröse az optimálisnak de töredéke a véletlen keresőnek főleg ha az átlagot nézzük. Tehát a szimulált lehűtés rossznak tűnhet a szélességihez képest de az időigénynél majd látni foglyuk, hogy fordul a kocka.

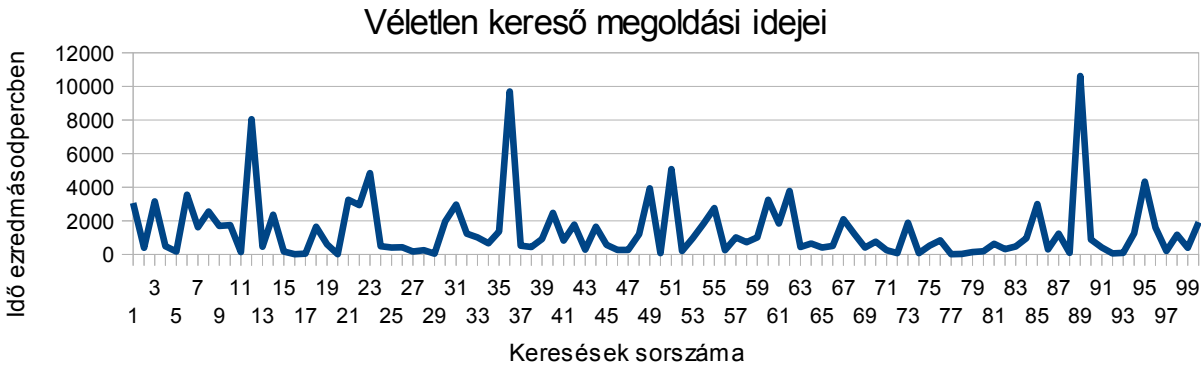
A **szélességi kereső** ha létezik megoldás akkor mindig a legrövidebb utat találja meg. A 10. ábrán is látható, hogy jelen esetben is a legrövidebb megoldást tárta fel ami most 25 állapotból áll a kezdő és célállapottal együtt.

A **hegymászó** ha megtalálná a megoldást akkor az optimális megoldás lehetne de a tesztelés során nem sikerült megoldást produkálnia így nem nevezhető optimálisnak.

3.3 Időigény

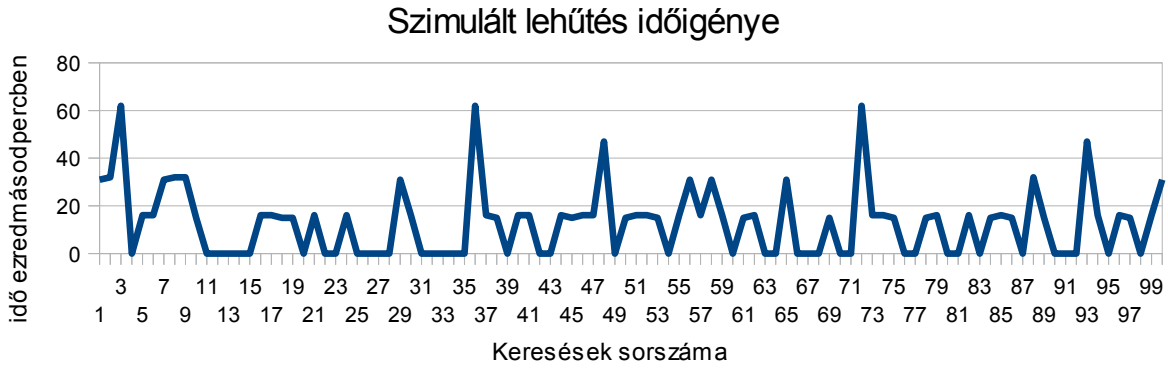
A **szélességi kereső** gyenge pontja itt lesz látható. A tesztelések során 10 és 15 másodperc alatta találta meg az egyébként optimális megoldást. A kereső működésében nincs véletlen így minden egye futtatáskor ugyan azt az időt kéne produkálnia. Ez így is lenne ha a gép processzora nem foglalkozna semmi mással. A determinisztikus működés miatt ebből nem is készítettem statisztikát mert az csak a gépemet jellemezte volna nem a keresőalgoritmust. Ezért fogadjuk el hogy körül belül 12,5 másodperc a futási ideje.

Már a **véletlen kereső** is jelentősen jobb eredményeket produkált.



18 ábra: Véletlen kereső megoldási idejei

A **szimulált lehűtés** kereső pedig még jobbakat.



19 ábra: Szimulált lehűtés megoldási idejei

Látható hogy már a véletlen keresőnek is 1 másodperc körül volt az átlag ami 10-ed része a szélességi idejének. A szimulált lehűtés idejét pedig nem is lehet a többihez hasonlítani. Az átlag 12,82 ezret másodperc. Ez század része a véletlen keresőnek és ezred része a szélességinek. Így már látható az igazi különbség. Lehet, hogy nem optimális a megoldása de gyorsan megvan.

Megoldási idő ezredmásodpercben

Szmulált		Véletlen
12,82	**ÁTLAG**	1415,81
0	**MIN**	15
62	**MAX**	10609

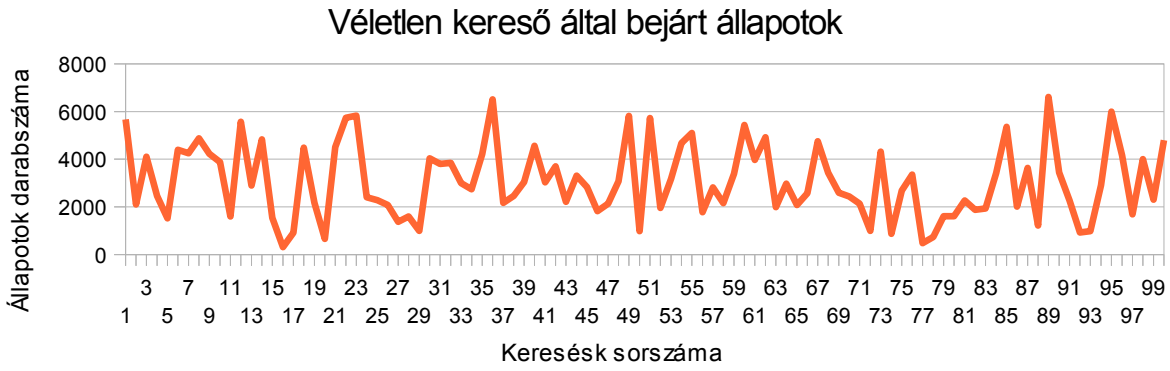
Vannak esetek amikor nem a célba vezető út az érdekes csak maga a lehetséges célállapotok. Ilyenkor jobb választás lehet ez az algoritmus.

A **hegymászót** most sem tudom külön értékelni hisz nem talált megoldást.

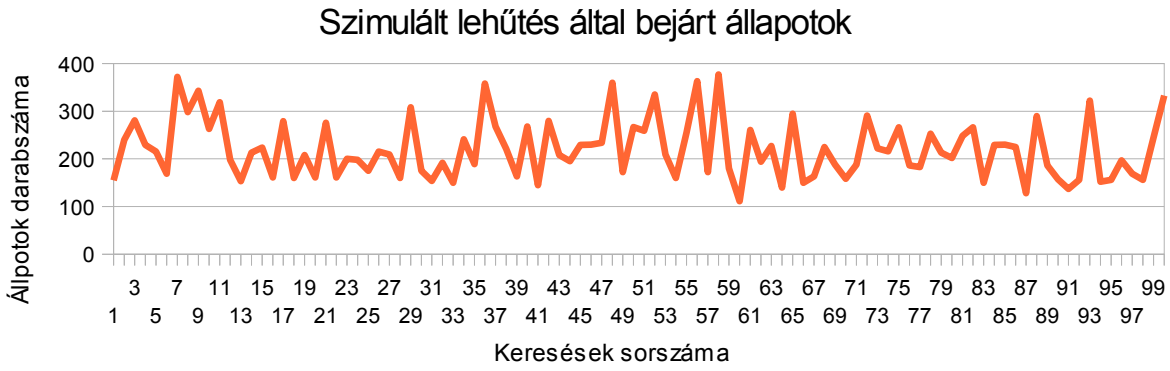
3.4 Tárigény

Az előzőhöz hasonló markáns különbségeket láthatunk majd most is.
A **szélességi** igazi hátránya nem az idő hanem a tárigény. Ha azt mondjuk hogy egy állapotnak van 'd' gyereke és 'l' hosszúságú a legrövidebb megoldás akkor
$1 + d + d^2 + d^3 + \ldots + d^{l+1} - d$.
Egy ilyen egyszerű feladatnál és egy ilyen egyszerű pályán mint ez is eltárol 9103 állapotot. Fentebb már láthattuk hogy összesen 22*21*20=9240 különböző állapot lehetséges. Tehát gyakorlatilag az összes állapotot tartalmazó keresőfát épít fel magának az algoritmus jelen esetben. Így kijelenthetjük hogy a tárigénye óriási.

Ezzel szemben a **véletlen** és a **szimulált lehűtés** keresői mindösszesen 1 darab állapotot tárolnak hiszen ők lokális keresőalgoritmusok. Viszont a két algoritmus között van egy nagy különbség abban melyik hány darab különböző állapotot állít elő a keresés során. Ez nem igazi tárigény hisze ezeket az állapotokat nem tároljuk egy időben de érdekes megvizsgálni.



20. ábra: Véletlen kereső



21 ábra: Szimulált lehűtés

Különböző állapotok

Szimulált		**Véletlen**
220,16	**ÁTLAG**	3068,15
111	**MIN**	309
377	**MAX**	6605

Láthatjuk, hogy a szimulált lehűtés itt is átlagosan tized annyi állapotot állított elő mint a véletlen kereső. Viszont a véletlen kereső legrosszabb estben sem állított elő annyi állapotot mint a szélességi kereső.

Összegzés

Láthatjuk tehát, hogy az állapottér méretei miatt igen is törekednünk kell az ügyes megoldásokra mert a véletlen kereső az életben igazi problémáknál nem használható. De ugyanakkor azt is láthatjuk, hogy igazodnunk kell a problémához. Először is gondot okozhat a heurisztika elkészítése. És ha ez nincs akkor máris nem használhatunk akármilyen keresőt. De ha heurisztikánk van is akkor is mérlegelnünk kell, hogy mi nekünk a fontos illetve mely lehetőségeink vannak korlátozva. Hiszen ha a tárkapacitásunk kicsi akkor a szélességi keresőt valószínűleg nem használhatjuk. Viszont ha van bőven tárkapacitásunk de sürget az idő akkor megint gond lehet a szélességi kereső használatával. Ha az optimalitás nem feltétlen fontos akkor ott van nekünk a szimulált lehűtés.

A szimulált lehűtést összegezve egy gyors hatékony megoldáskereső kis tárigénnyel. Hátránya ugyan, hogy optimalizálni kell külön az egyes feladatokhoz az időfüggvényét ami időigényes lehet. De ha ezt a problémát megoldjuk akkor egy igazán jó keresőt kapunk.

Ideális az olyan feladatok esetén ahol a célba vezető út nem fontos csak a célállapotokat keressük.

A szimulált lehűtés először a 80-as években terjedt el és azóta széles körben használják ipari termelés ütemezésre és nagyobb volumenű optimalizációs feladatok feladatokra.

Gyönyörű példája az emberi leleményességnek hiszen egy kohászati eljárás adta hozzá az ötletet. Arra is szép példa hogy mennyivel többet ér ha gondolkodik az ember nem pedig nyers erővel akar nekilátni a probléma megoldásának.

A heurisztikus keresőknél fontos a jó heurisztika is. Hiába alkalmaz jó szisztémát a kereső ha a heurisztikánk nem tökéletes akkor a keresés sem lesz az.

A keresőalgoritmusok kutatása és fejlesztése biztos, hogy sok dolgozat témája lesz még hisz folyamatosan változnak, javulna a technikai feltételek de ezzel együtt az emberek is egyre nagyobb feladatokat találnak ki a gépek számára. Mindig kicsivel nagyobbat mit amit a gép meg tudna oldani így aztán a folyamatosan szükség van az új ötletekre, újításokra ezen a téren is csakúgy mint az élet sok más területén.

Mellékelt program

A szakdolgozathoz tartozó CD-n megtalálható a mesterséges intelligencia gyakorlaton megírt osztályok, keresők kibővítve az általam tárgyalt keresőkkel és a Threesom játékkal. Tartalmazza még eme szakdolgozatot .pdf formátumban és két táblázatot amiben a keresők mérési eredményei vannak a grafikonokkal együtt.

Köszönetnyilvánítás

Szeretnék köszönetet mondani Kósa Márk Tanár Úrnak aki tanácsaival és iránymutatásával segítette eme szakdolgozat megszületését.
Köszönöm a segítséget és biztatást a csoporttársaimnak Csillag Péternek, Horváth Attilának , Nagy Istvánnak, Szauervein Szabolcsnak, Tanyi Attilának és végül de nem utolsó sorban Zelenák Zoltánnak
Szeretném még megköszönni a támogatást szüleimnek akik nélkül nem tölthettem volna el ezeket a boldog éveket az egyetemen.

Irodalomjegyzék

[1] Dr. Várterész Magdolna, Mesterséges Intelligencia 1 előadások (fóliák)
Lásd: http://www.inf.unideb.hu/~varteres/mi1folia/
[2] Mesterséges Intelligencia (Szerk. Futó Iván), Aula Kiadó, 1999
[3] Stuart Russell, Peter Norvig. Mesterséges intelligencia modern megközelítésben. (2. kiadás), Panem Kiadó Kft., 2005.
[4] A Mesterséges Intelligencia 1 tágy honlapján elérhető forráskódok
Lásd: it.inf.unideb.hu/mestint